공학석사학위논문

비압축성 나비에-스토크스 방정식의 GPGPU 구현을 위한 semi-implicit 레드/블랙 solver

Semi-Implicit Red/Black Solver for Incompressible Navier-Stokes Equations on GPGPU

2013년 8월

서울대학교 대학원
전기컴퓨터공학부
Dmitry Timofeev

Abstract

Semi-Implicit Red/Black Solver for Incompressible Navier-Stokes Equations on GPGPU

School of Electrical Engineering and Computer Science
The Graduate School
Seoul National University

A novel Semi-Implicit Bicolor Navier-Stokes solver, inspired by red/black scheme for Poisson equation, was developed for simulating incompressible fluids on GPGPU. The solver was shown to be beneficial for simulating highly unsteady flows. Also, it allows more freedom in choice of time step because of the relaxed stability condition. Spatial discretization uses staggered grid and approximates convective terms with WENO scheme to capture the effects of convection-dominated flows. The computation is dominated by the Poisson solver, which uses matrix-free Preconditioned Conjugate Gradient algorithm. Several known matrix-free preconditioners were considered. Red/Black SSOR was found to be the most suitable matrix-free preconditioner for PCG, even though it is usually used as a main solver. A mixed precision modification with explicit restarts of the PCG was shown to be slightly beneficial in unsteady region, while giving a negative effect for

steady flow. Sensitivity analysis of the solver was carried for further performance optimization.

Keywords : Incompressible Navier-Stokes equations, Red/Black scheme, unsteady flow, staggered grid, WENO, Poisson equation, Red/Black SSOR, mixed precision

Student Number : 2011-24073

Contents

제 1 장

Introduction

Numerical solution of incompressible Navier-Stokes equations is a very computationally expensive task even in two-dimensional case. In fact, during past decades high resolution fluid simulations were perfomed on large clusters and supercomputers only. However, in last several years general purpose graphic cards (GPGPUs) became more recognized by the scientific community as a powerfull computational tool. Graphic processors allow to perform same calculations on small cost-effective workstations.

However, GPGPU architecture is beneficial only for highly parallel computations. This restriction, along with computational accuracy and stability issues of known numerical algorithms limits the choice of solvers. In order to ensure highly parallel nature of the algorithm, which is necessary for GPU computing, explicit schemes are usually chosen as basic time discretization. However, this choice results in restrictions on time step and, as a consequence, in high overall time cost. This work presents two approaches to reduce overall computational cost.

First, an ordinary explicit time marching scheme is considered. Most of the time in such schemes is spent on solving the Poisson problem for pressure (from 97% in rapidly changing phase to 45% close to steady state solution). Therefore, optimization of Poisson solver will have a good impact on

overall performance. Matrix-free Preconditioned Conjugate Gradient (PCG) algorithm is considered in this work. Absence of the explicit problem matrix in the main solver limited the choice of preconditioner to matrix-free ones. Red/Black Successive Over-Relaxation (RB-SSOR) preconditioner showed the best performance on test problems, compared to other known matrix-free preconditioning techniques, including recently published.

Second, the marching scheme was considered for optimization. Inspired by Red/Black SSOR, a bicolor semi-implicit time marching algorithm for Navier-Stokes was developed. Semi-implicit character of the solver allows a weaker restriction on time step. This results in smaller overall cost for unsteady flows and could also be beneficial itself, since the engineer has more freedom in choosing the time step.

제 2 장

Related Work

Fluid simulations on graphics hardware have long been of interest mainly for computer graphic applications, where accuracy is not essential, but speed is. In this field, flow simulations using the method presented by Stam [1] are very popular. It allows large time stepping for solving Navier-Stokes equations with excelent stability. Unfortunately, the method is not accurate enough for engineering applications, but does capture the fluid motion with good visual appearence. Harris et al. [2] performed visually-realistic cloud simulation using Stam's method. Liu et al. [3] performed flow calculations over the obstacles, e.g. flow over a city, using the same method. Goal of all these authors was to have a real-time solver along with visualization running on GPU.

With appearence of double precision calculation capabilities on GPGPUs, some papers with engineering CFD solvers on GPGPU came out. Bradvic and Pullan [4] implemented a finite volume solver for compressible Euler equations, modeling inviscid fluid. They achieved speed-ups up to 29 compared to their CPU solver. Elsen et al. [5] developed a finite difference solver for compressible Euler equations. They saw speedup of up to 17 in simulations of flow over the hypersonic vehicle.

Most papers on simulating viscous fluids on GPU, use projection method

with Euler marching. Liu et al. [6] implemented an Euler marching algorithm for incompressible Navier-Stokes problem. They used a Red/Black SOR for solving the Poisson equation on every time step, seeing up to second order speedup (compared to sequential solver) in simulations of lid-driven cavity problem with Reynolds number $Re = 100$. Griebel and Zaspel [7] developed a multi-GPU implementation of their CPU based solver for two-phase incompressible Navier-Stokes problem. They used Euler time marching and a Conjugate Gradient solver with Jacobi preconditioning to handle the Poisson problem. Griebel and Zaspel reached speed-up of almost 70 compared to their serial CPU code. Thibault and Senocak [8] developed a multi-GPU solver with Jacobi method for pressure problem and Euler marching for time stepping. Projection method with Euler marching was implemented in this work as a reference algorithm for comparison.

Since the Poisson problem for pressure alone is an important part of CFD algorithms, it was considered for solving on GPU by some authors. Here we will mention some of these works. Konstantinidis and Cotronis described their Red/Black SOR solver in [9]. They saw a speedup of 10 compared to sequential implementation of the same solver. Ament et al. [10] developed an approximate inverse preconditioner for Conjugate Gradient solver of Poisson problem on multi-GPU platform. This heuristic preconditioner was later obtained theoretically by Helfenstein and Koko in [11]. This Incomplete Poisson Preconditioner for CG method by Ament et al. was implemented for comparison in this research.

제 3 장

CUDA computing model

In CUDA computing model functions, called kernels, are executed on a computational grid. Computational grid consists of some amount of blocks, which in turn consist of threads (fig. 1).

Each block resides on a separate streaming multiprocessor (SM). SMs have some amount of on-chip shared memory, which is available for all threads of the same block. Also, individual thread gets some register memory, which is unaccessable for other threads. Synchronization mechanism for block of threads is available. On the contrary, there is practically no global synchronization mechanism for different blocks, apart from start of a new kernel.

Before the kernel launch, all the data is loaded to the global memory (DRAM). During the kernel execution, data should be fetched to the SM. Unfortunately, access to the global memory has a very high latency, so the values to be reused should be saved in shared memory of the block or registers of individual threads, depending on the access patterns.

Parallelerism analysis and data access patterns, along with specific implementation details for Poisson and full Navier-Stokes solvers could be found in subsections 5.5 and 6.4 correspondingly, after the algorithms are discussed.

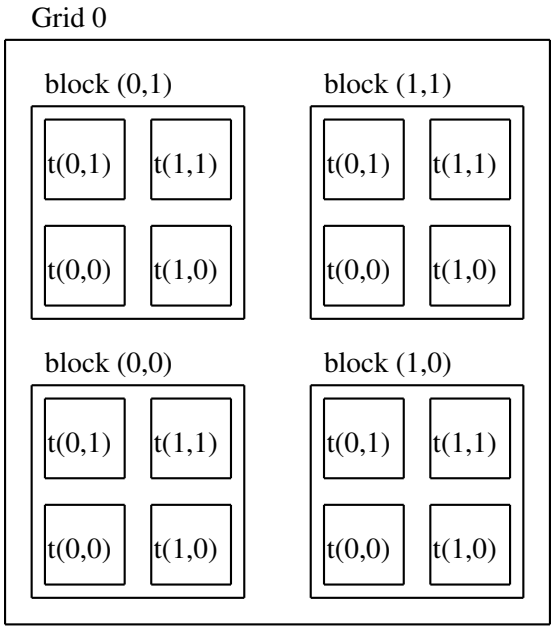

Fig. 1: Computational Grid

제 4 장

Navier-Stokes and Poisson equations

Incompressible Navier-Stokes (INS) equations describe a conservation law of fluid's velocities. Given the physical properties, initial and boundary conditions (IC and BCs), INS equations desribe the evolution of velocities in the domain. In contrast to graphic fluid simulations (which usually assume inviscid fluid), engineering and scientific applications demand modeling of incomressible fluids with non-zero viscosity, also called Newtonian fluids. The mathematical formulation of Navier-Stokes problem is given below.

4.1 Navier-Stokes Equations

Let Ω be the simulation domain with boundary Γ and $T > 0$ the simulation time. Then, governing equations of unsteady Newtonian flow with Dirichlet boundary conditions are written in pimitive variables as:

$$\begin{cases} \dfrac{\partial \mathbf{u}}{\partial t} + \nabla p = -(\mathbf{u}\cdot\nabla\mathbf{u}) + \mathbf{g} + \dfrac{1}{Re}\Delta\mathbf{u} & \\ \nabla\cdot\mathbf{u} = 0 & \\ \mathbf{u}(\mathbf{x},t) = \mathbf{U}(\mathbf{x},t), & \mathbf{x}\in\Gamma \\ \mathbf{u}(\mathbf{x},0) = \mathbf{U}_0(\mathbf{x}), & \mathbf{x}\in\Omega \end{cases} \tag{4.1}$$

Here, $\mathbf{u}$ - fluid velocities, p - hydrostatic pressure, $Re = \frac{\rho V L}{\mu}$ - Reynolds

number for given density ρ, characteristic length L and velocity V and dynamic viscosity μ. First equation describes velocity field evolution in time and is often refered as Navier-Stokes equation. Second equation in (4.1) states both mass conservation law and incompressibility. Two other equalities give boundary (BC) and initial (IC) conditions respectively.

This system contains equations, coupling the velocities and pressure. In this form a system of equations is hard to solve. A commonly used approach to separate the variables is projection method, like Helmholtz-Hodge decomposition or Chorin's projection method [12].

4.2 Projection method

Main idea of the projection methods is the following: on every time step, consider an intermidiate solution u^*, which satisfies only the evolutionary equation without the pressure. Such a solution will be not divergence-free, which is demanded by conservation law. Based on it, formulate the problem for pressure, such that the solution on the next time step $u^{n+1} = u^* - \nabla p$ will be divergence-free. The projection method [13], which is used in current work is sketched below.

Denote right-hand side of the first equation in (4.1) as $\mathbf{f}$:

$$\mathbf{f} = -(\mathbf{u} \cdot \nabla \mathbf{u}) + \frac{1}{Re}\Delta \mathbf{u} \tag{4.2}$$

Then the governing equation becomes

$$\frac{\partial \mathbf{u}}{\partial t} = -\nabla p + \mathbf{f} \tag{4.3}$$

Applying divergence operator $\nabla\cdot$ to both sides of equation (4.3) and taking into account that $\nabla \cdot \dfrac{\partial \mathbf{u}}{\partial t} = 0$ (due to incompressibility law), we obtain an equation for pressure only:

$$\nabla^2 p = \Delta p = \nabla \cdot \mathbf{f} \tag{4.4}$$

This is a Poisson problem for pressure. In order to solve (4.4), we need to specify appropriate BCs for pressure. In order to get these, mutiply (4.3) by an outer normal vector $\mathbf{n}$ on the boundary:

$$\nabla p \cdot \mathbf{n} = \frac{\partial p}{\partial \mathbf{n}} = \mathbf{f} \cdot \mathbf{n} - \frac{\partial \mathbf{u}}{\partial t} \cdot \mathbf{n} \tag{4.5}$$

The right-hand side of Poisson problem (4.4-4.5) is changing on every marching step, so it should be solved as many times, as there will be time steps.

The resulting time marching process is done as following:

1. Compute $\mathbf{f}$, with (4.2), using the velocity field from the previous time step $\mathbf{u}^n$

2. Solve the Poisson problem with Neumann BC (4.4-4.5) for pressure

3. Compute divergence-free velocity field on the next time step from

$$\frac{\partial \mathbf{u}^{n+1}}{\partial t} = \mathbf{f} - \nabla p \tag{4.6}$$

using some time discretization method.

제 5 장

Poisson problem

Consider an isolated Poisson problem with Neumann type boundary condition

$$\Delta p = \varphi, \frac{\partial p}{\partial \mathbf{n}} = \gamma \tag{5.1}$$

where Δ denotes a Laplacian, p is a function in closed domain $\overline{\Omega} = \Omega \cup \Gamma$, φ and γ are functions in Ω and Γ respectively.

5.1 Discretization

Unfortunately, analytical solution of the problem (5.1) is feasible only for simplified domains and specific right-hand side functions φ. Instead, a finite number of grid points is introduced to approximate the derivatives with finite differences. We will use an equidistant structured grid, which is a widely used procedure for Poisson problem. Denote solution and right-hand side values at the point (x_i, y_j, z_k) as p_{ijk} and φ_{ijk}. Then, the discrete approximation of (5.1) in three-dimensional case for an internal point is

$$\Delta p \approx \frac{p_{i+1} + p_{i-1} + p_{j-1} + p_{j+1} + p_{k-1} + p_{k+1} - 6p}{h^2} = \varphi \tag{5.2}$$

For convinience, unmodified indices corresponding to the node (x_i, y_j, z_k) are omitted in (5.2).

System with equations for all the grid points should be solved. Since the demands for accuracy in engineering and scientific applications are usually quite high, such systems could be very large.

5.2 The PCG Algorithm

The Preconditioned Conjugate Gradient (PCG) method is a widely used iterative algorithm for solving symmetric positive definite linear systems. The matrix of the problem (5.2) is well studied and satisfies all the constraints of convergence for PCG.

Provided input vectors of initial guess p_0 and right-hand side φ, the solution of $Ap = \varphi$ with the PCG algorithm and a preconditioner M takes a form of

$$r_0 = \varphi - Ap_0 \qquad h = M^{-1}r_0 \qquad d_0 = h \qquad r_{old} = (r_0, h) \qquad l = 1 \quad (5.3)$$

`While` $l < l_{max}$ and $r_{old} > \varepsilon$ `do`

$$\begin{aligned} t &= Ad_{l-1} & \alpha &= \frac{r_{old}}{(d_{l-1}, t)} \\ r_l &= r_{l-1} - \alpha t & p_l &= p_{l-1} + \alpha d_{l-1} \\ h &= M^{-1}r_l & r_{new} &= (r_l, h) \\ \beta &= \frac{r_{new}}{r_{old}} & d_l &= h + \beta d_{l-1} \\ r_{old} &= r_{new} & l &= l + 1 \end{aligned} \quad (5.4)$$

Note that it is possible to avoid keeping the matrices A and M^{-1} explicitely in the memory - if multiplication of an arbitrary vector by these matrices can be done with some point operator (applying some operator node by node). Such matrix-free implementation will limit storage and memory access demands to only five one-dimensional vectors.

For more background, theory and derivation of PCG algorithm and other contemporary Krylov subspace methods, refer to a book by Saad [14].

5.3 Preconditioning

Roughly speaking, a preconditioner is any form of implicit or explicit modification of an original linear system which makes it "easier" to solve by a given iterative method. For example, scaling all rows of a linear system to make the diagonal elements equal to one is an explicit form of preconditioning. The resulting system can be solved by a Krylov subspace method, such as CG, and may require fewer steps to converge than with the original system (although this is not guaranteed).

The right preconditioned linear system $Ax = b$ takes the form:

$$AM^{-1}u = b, \quad u = Mx \tag{5.5}$$

The matrix M is the preconditioning matrix. It should be close to A, i.e. $M \approx A$ and the preconditioning operation $M^{-1}v$ should be easy to apply for an arbitrary vector v.

The convergence rate of iterative methods is highly dependent on the quality of the preconditioner used. Computational complexities of different

types of preconditioning operations are different. Moreover, their performance varies significantly on different computing platforms.

CG method's convergence rate depends mainly on two parameters: size of the system and its condition number. For symmetric positive definite matrices, the condition number κ is defined as

$$\kappa(A) = \frac{\lambda_{max}}{\lambda_{min}} \tag{5.6}$$

with maximum and minimum eigenvalues $\lambda_{max}(A)$ and $\lambda_{min}(A)$. Note, that the identity matrix has a condition number $\kappa(I) = 1$.

The closer system's condition number is to that of identity matrix, the faster CG method will converge. The objective of preconditioning is to transform the original system into an equivalent system with the same solution, but a lower condition number.

However, the computational overhead of applying the preconditioner must not cancel out the benefit of fewer iterations. Also, increased memory demands (for storing the preconditioning matrix) should be taken into account.

5.4 Existing Matrix-Free Preconditioners for GPU

Since large Poisson problems are very common in scientific applications, so are matrix-free implementations of CG for them. However, most of recent works on numerical linear algebra on GPU (refer to, for example, [15, 16, 17, 18]) consider explicit storage of the problem matrix and preconditioner.

Usage of such preconditioners may be not possible with the matrix-free CG method.

Only matrix-free preconditioners for GPU will be discussed in this work. For other preconditioning and implementation schemes the reader can refer to the above mentioned papers [15, 16, 17, 18] and other recent works on efficient GPGPU computing.

We will take advantage of matrix notation below where it is useful. For a symmetric matrix of system (5.2), the following decomposition is used:

$$A = L + D + L^T \tag{5.7}$$

where D is a diagonal matrix of diagonal elements of A and L is a lower triangular part of A.

5.4.1 Jacobi Preconditioning

Jacobi preconditioner basically applies single iteration of Jacobi method. The idea of point-Jacobi iteration is to cancel out each component of the residual vector individually. Thus, for i-th component:

$$(b - Ax)_i = 0 \tag{5.8}$$

Gathering equations of the form (5.8) in a vector form of iterative technique, one can obtain

$$x^{k+1} = x^k + D^{-1}(b - Ax^k) \tag{5.9}$$

Or, componentwise

$$x_i^{k+1} = x^k + \frac{1}{a_{ii}}(b_i - (Ax^k)_i) \tag{5.10}$$

If the matrix-free operator standing for applying matrix A is available, Jacobi iteration could be applied component-wise in an independent manner. This fact offers a full parallelerism on GPU: each thread could apply this operation to every node, with no communication overhead.

However, the information about other updated nodes is not used. This results in just a minor decrease in number of iteration for convergence, which could possibly be outweighted by increased computational effrots.

5.4.2 Symmetric Successive Over-Relaxation

Gauss-Seidel (G-S) method in a classic sequential formulation corrects the i-th component of the current approximate solution, in the order $i = 1, 2, \ldots, N$, again, to annihilate the i-th component of the residual. However, this time the approximate solution is updated immediately after the new component is determined. This difference allows to take advantage of newly refined solution on previous nodes. Practically this implies solution of lower triangular system

$$(D+L)x^{k+1} = (D+L)x^k + (b - Ax^k) \tag{5.11}$$

Successive Over-Relaxation (SOR) modification of G-S introduces relaxation parameter to system (5.11) for better convergence rate:

$$(D+\omega L)x^{k+1} = (D+\omega L)x^k + \omega(b - Ax^k) \tag{5.12}$$

Same process could be applied in opposite direction - in the backward order $i = N, N-1, \ldots, 1$, resulting in solution of upper triangular system

$$(D + \omega L^T)x^{k+1} = (D + \omega L^T)x^k + \omega(b - Ax^k) \tag{5.13}$$

This procedure is called backward SOR.

Symmetric SOR (SSOR) includes a forward SOR sweep (5.12), followed by a backward sweep (5.13). SSOR is a widely used preconditioning method in sequential applications, since it is relatively easy to implement and gives good convergence properties.

However, solving the triangular system is a sequential operation, which could not be effectively mapped on any parallel architectures without loosing some of performance advantages.

In the following, we will discuss some modifications of SSOR, designed for parallel architectures and present red-black SSOR for parallel architectures.

5.4.3 Block-Grained SSOR

The most natural design idea of SSOR metod modification is to divide the matrix in blocks (see fig. 2) and apply the algorithm to each block independently. This will result in some loss of information - since not all the nodes are updated with the most recent information. However, the approach will still be advantageous in comparison with Jacobi or absence of preconditioning.

This strategy is often used in practice for multicore CPU implemen-

tations. Since the number of computing cores is small, so is the number of blocks. For big block sizes, impact of Block-Grained SSOR (BG-SSOR) on CG convergence rate is still quite high.

GPUs offer a massive fine-grained parallelerism, but the computing power of a single thread is quite low. While BG-SSOR allowes certain parallelerism in working with separate blocks of the matrix, triangular system of each block still has to be solved sequentially by a single thread.

To make the computation on GPU more effective, we will have to divide the matrix in smaller blocks, which will result in a much slower convergence. Resulting in just a minor decrease in number of iterations, this gain of preconditioning could be outweighted by a highly increased computational effort, since practically all the calculations in a block are done by a single thread.

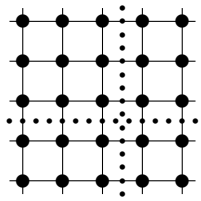

Fig. 2: Breaking the mesh into blocks for independent SSOR preconditioning

5.4.4 Approximate Inverse SSOR

SSOR preconditioning matrix M could be expressed in terms of splitting (5.7) of A:

$$M(\omega) = \frac{1}{2-\omega}(\frac{1}{\omega}D + L)(\frac{1}{\omega}D)^{-1}(\frac{1}{\omega}D + L)^T \tag{5.14}$$

Unfortunately, PCG algorithm requires knowledge of inverse of matrix M, computing which is not feasible for arbitrary boundary conditions. However, approximate inverse of this matrix could be obtained. This approach is used in recent paper [10] by Ament et.al.

The Incomplete Poisson (IP) preconditioner, obtained by Ament et.al. in [10] by heuristic approach could be expressed as

$$M^{-1} \approx \bar{M} = (I - LD^{-1})(I - D^{-1}L^T) \tag{5.15}$$

In order to reduce the computational cost, IP preconditioner could be computed with enforced sparsity pattern of matrix A. Resulting loss in convergence rate was found to be rather small in comparison to decrease in computational effort [10].

For both preconditioners explicit matrix-free formulas could be obtained - see [10] for those of IP preconditioner.

5.4.5 Red/Black SSOR

Red/Black SSOR (RB-SSOR) is usually used as a main solver itself. However, since it is a modification of SSOR method, carrying most of its good convergence properties, it could be used as preconditioner as well.

Red-Black scheme exploits the properties of the grid used for discretization (5.2). As the name suggests, this grid could be colored in red and black (fig. 3), such that no adjacent nodes have same color.

This means, that nodes of the same color are not connected, so their corresponding values do not appear in same equation. Therefore, nodes of the same color could be updated simultaneously.

The idea is to simultaneously update red nodes first (which depend on black nodes only) and then do the simmilar operation for black nodes (depending only on red ones) using the most recently updated values. This will complete the forward red-black SOR sweep.

To do the backward sweep, we need to update the nodes in a reverse order: all the black ones simultaneously first and the red ones afterwards. Mathematically, this is simmilar to reordering the matrix by colors (fig. 4) before applying the original sequential SSOR.

However, there is no need to reorder the matrix explicitly - we can process the nodes with $(i+j+k)$ even as red and $(i+j+k)$ odd as black ones.

If the original sequential SSOR method was applied to color-ordered

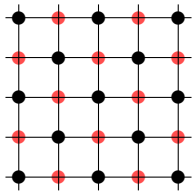

Fig. 3: Red/Black colored grid

red-black matrix - the resulting convergence rate will be simmilar, so there is practically no loss in mapping the algorithm to the parallel architecture. Decrease in convergence rate compared to original sequential SSOR is only due to this virtual reordering and is very small.

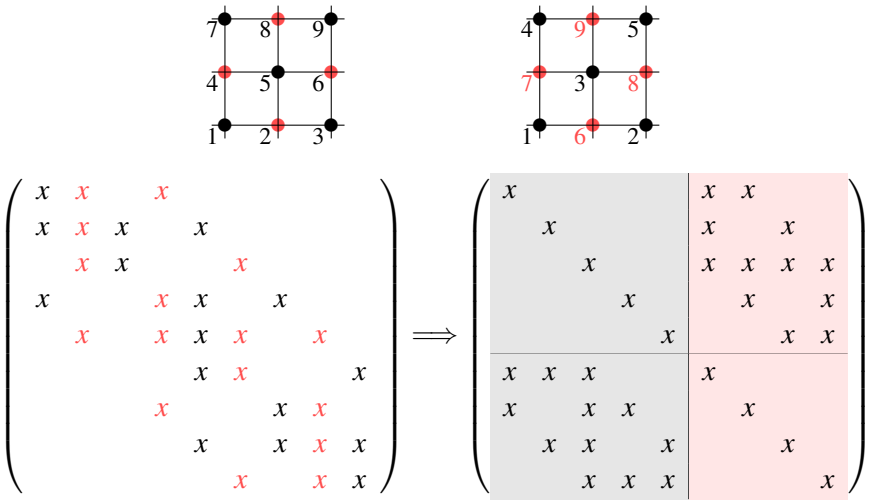

Fig. 4: Red/Black reordering of the grid nodes and corresponding matrix reordering

5.5 Implementation Details

Recall the PCG algorithm 1 for the Poisson problem. Examining the main loop, one can mention the following:

Algorithm 1 PCG

1: $r_0 = \varphi - Ap_0 \qquad h = M^{-1}r_0 \qquad d_0 = h \qquad r_{old} = (r_0, h) \qquad l = 1$

2: **while** $l < l_{max}$ and $r_{old} > \varepsilon$ **do**

3: $\quad t = Ad_{l-1}$

4: $\quad \alpha = \dfrac{r_{old}}{(d_{l-1}, t)}$

5: $\quad r_l = r_{l-1} - \alpha t$

6: $\quad p_l = p_{l-1} + \alpha d_{l-1}$

7: $\quad h = M^{-1}r_l$

8: $\quad r_{new} = (r_l, h)$

9: $\quad \beta = \dfrac{r_{new}}{r_{old}}$

10: $\quad d_l = h + \beta d_{l-1}$

11: $\quad r_{old} = r_{new}$

12: $\quad l = l + 1$

13: **end while**

- Result of step 3 is used for dot product in 4 and has to be computed first. It should be implemented in a separate kernel
- Dot product on lines 4 and 9 is a reduction operation
- Residual (5) and solution (3) are updated independently, which could be done in the same kernel
- Preconditioning (7) demands a separate kernel, since the result is used immidiately after. Jacobi and IP preconditioners are implemented in a single kernel, while Red/Black SSOR needs 4 kernel calls - one for each color update

- The residual value r_{old} has to be transfered to the host side after each loop for checking the convergence criteria. However, this could as well be done in asynchronous manner: it could result in a few extra iterations, but the computation will not be stalled between iterations

Taking into account all the above, algorithm could be rewritten (each line is a kernel call):

Algorithm 2 PCG

1: $r_0 = \varphi - Ap_0$

2: $h = M^{-1}r_0$, $d_0 = h$

3: Compute $r_{old} = (r_0, h)$ with parallel reduction, set $l = 1$

4: **while** $l < l_{max}$ and $r_{old} > \varepsilon$ **do**

5: $t = Ad_{l-1}$

6: Compute (d_{l-1}, t) with parallel reduction

7: Compute $\alpha = \dfrac{r_{old}}{(d_{l-1}, t)}$, $r_l = r_{l-1} - \alpha t$ and $p_l = p_{l-1} + \alpha d_{l-1}$

8: Apply preconditioner $h = M^{-1}r_l$

9: Compute $r_{new} = (r_l, h)$ with parallel reduction

10: Get $\beta = \dfrac{r_{new}}{r_{old}}$ and compute $d_l = h + \beta d_{l-1}$, set $r_{old} = r_{new}$, $l = l + 1$

11: **end while**

There are 3 types of kernels in PCG algorithm 2:

- Vector-vector summ (updating p, r and d)
- Two-phase reduction for dot product
- Matrix-vector operation (application of problem operator and precon-

ditioning)

Vector-vector summation doesn't require loading the values to shared memory: coordinate-wise data is loaded, processed in a single operation and not reused again. These are done using global memory only.

Two-phase parallel reduction is used for dot product. It makes use of shared memory and is a widely known procedure. On the first phase, reduction is carried by blocks. On the second phase, the resulting vector is reduced by a single block.

Both matrix-vector operations are implemented with point operators, in a matrix-free manner. Problem operator approximates Laplacian Δp and needs values of p at the corresponding node (i, j) an its neighbours $(i \pm 1, j \pm 1)$. These values are also used for computations in other nodes. Global memory has high latency, so it is necessary to store these values in shared memory for reusing. On launch of matrix-vector kernel each thread loads value at corresponding node. Threads on the block boundary also load neighbouring data from the block halo.

To ensure coalescing of the memory access, data is organized according to the computational grid. Values are stored in one-dimensional array with natural thread indexing. For example, to access the cell to be processed by thread `(th.x,th.y)` in block `(b.x,b.y)` of grid `(g.x,g.y)`, we need to use its global index:

$$\texttt{gid} = \texttt{t.x} + \texttt{t.y} \cdot \texttt{bDim.x} + (\texttt{b.x} + \texttt{b.y} \cdot \texttt{g.x}) \cdot \texttt{bDim.x} \cdot \texttt{bDim.y} \cdot \texttt{g.x} \cdot \texttt{g.y}$$

Since the data is organized according to natural thread order, memory access is coalesced.

Preconditioner follows the grid pattern of the problem operator, so data access requirements are simmilar.

5.6 Performance of Poisson Solver

Performance of PCG with described preconditioning techniques was measured on Poisson problem with Dirichlet boundary conditions for various domain sizes. Sequential algorithm with classic SSOR was implemented on CPU to compare platform-independent covergence properties of preconditioners. PCG with Block-Grained SSOR was tested only on problem with dimensions $64 \times 64 \times 64$ because of its poor performance.

5.6.1 Iteration count and convergence gain

Total iteration count and covergence gain of considered preconditioners are given on fig. 5-6.

As expected, Jacobi preconditioner showed only minor gain in convergence rate, as well as Block-Grained SSOR. Implementation of Incomplete Poisson preconditioner gave a significant decrease in total iteration count - up to 39%, while Red-Black SSOR was close to performance of state-of-the-art SSOR preconditioning with up to 65% increase of convergence rate.

5.6.2 Iteration cost

Computed cost per iteration is given on fig. 7.

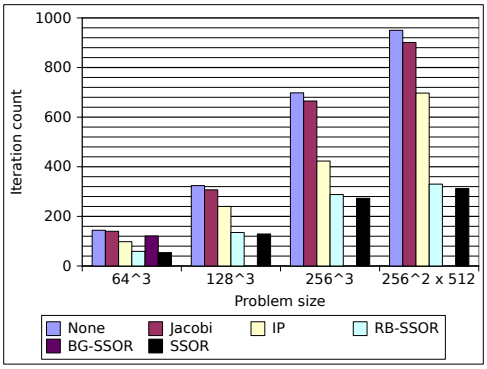

Fig. 5: Comparison of total iteration count for considered preconditioners on a reference problem

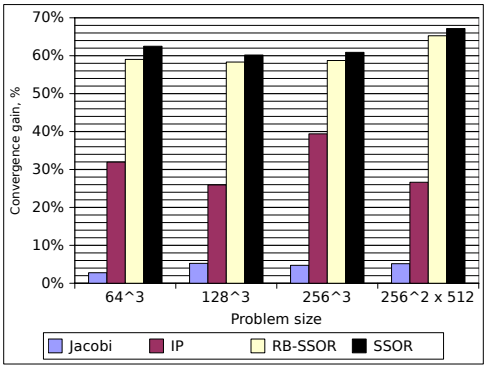

Fig. 6: Convergence gain of considered preconditioners as compared to absence of preconditioning

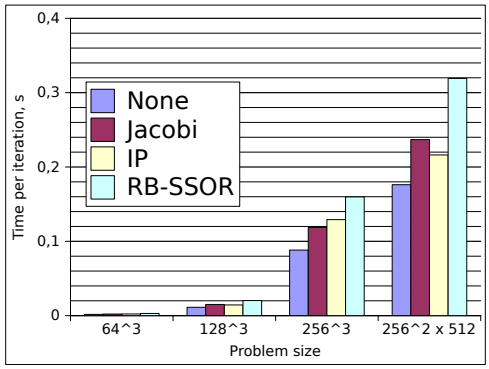

Fig. 7: Comparison of time cost per iteration

In most tests increase of iteration cost of IP preconditioner was almost simmilar to that of Jacobi. This is expcted, since the computational cost of IP is close to cost of application of the point-wise operator A.

The Red-Black SSOR increased the iteration cost twice more than Jacobi or IP. Although 4 kernel launches and more global memory accesses are performed, it needs twice less threads to perform the preconditioning and most of the global memory accessing latency is hidden. As a result, increase of iteration cost is not much higher compared to absence of preconditioning, while gain in convergence is considerable.

5.6.3 Overall performance

Finally, overall performance of different preconditioners is compared on fig. 8.

Block-Grained SSOR showed very poor performance even on a relatively small problem. This is due to extremely high computational complex-

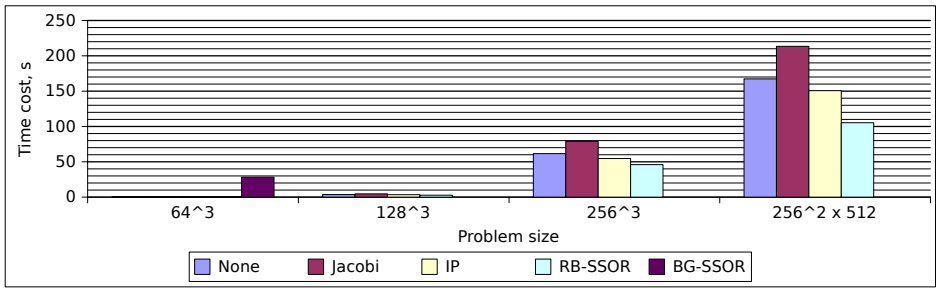

Fig. 8: Comparison of overall computing time

ity and only a minor increase in convergence rate.

Jacobi preconditioning showed better results, but additional computational complexity still outweighted small convergence gain.

In fact, only IP and RB-SSOR preconditioners turned out to be beneficial in overall computation. Even though RB-SSOR is about twice more costly to apply than IP, it outperforms the latter thanks to a significantly higher convergence rate.

Overall performance gain (fig. 9) of RB-SSOR reached 37%, while that of IP was getting up to 11%.

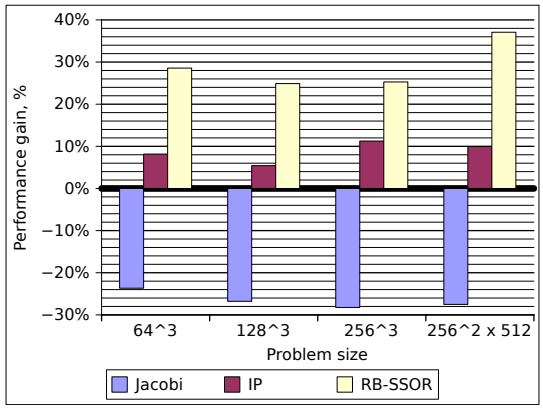

Fig. 9: Comparison of overall performance gain

제 6 장

Navier-Stokes Problem Approach

Approaching the INS Problem with the projection method described earlier in section 4 needs three issues to be resolved. First, we need to choose the spatial discretization technique for right-hand side **f** in (4.2) to have both a good approximation and capture convection-dominated flows. Second, Poisson problem for pressure has to be set up consistently. Third, appropriate time marching procedure should be chosen.

6.1 Spatial discretization

Let us consider the numerical treatment of momentum equation. We will use a staggered grid for discretization of velocities. Staggered grids are formed the following way: the cell centers $\mathbf{x}_{i,j,k}$ correspond to pressure nodes, whereas the cell-face centers $\mathbf{x}_{i+1/2,j,k}$, $\mathbf{x}_{i,j+1/2,k}$ and $\mathbf{x}_{i,j,k+1/2}$ give the velocity nodes for u_1, u_2 and u_3 respectively. Fig. 10 gives an example of two-dimensional staggered grid cell.

For diffusive terms we will use central second order differences. This will be simmilar grid operator as the one used for discretization of Poisson equation.

However, to properly approximate convection dominated flows, we cannot use central differences for convective terms. Instead, a third-order

WENO (Weighted Essentially Non-Oscillatory) scheme was used (for more information on WENO scheme refer, for example, to pioneering paper by Liu et.al. [19]). Constructed 3-rd order WENO scheme is sketched below for approximation of $\frac{\partial u}{\partial x}$.

Consider three point stencil (fig. 11). We will approximate the derivative by a weighted combination of forward difference u_x and backward difference $u_{\bar{x}}$. Weights are computed based on nonlinear local smoothness indicators:

$$\begin{aligned}
&\frac{\partial u}{\partial x} \approx w_1 u_{\bar{x}} + w_2 u_x \\
&w_k = \frac{\widetilde{w}_k}{\widetilde{w}_1 + \widetilde{w}_2} \\
&\widetilde{w}_k = \frac{1}{\varepsilon + \beta_k^2} \\
&\beta_1 = (u - u_{-1})^2, \qquad \beta_2 = (u_{+1} - u)^2
\end{aligned} \tag{6.1}$$

Here, β_k are nonlinear smoothness indicators and ε - some small number (in this work, $\varepsilon = 10^{-6}$ was chosen), which is introduced to avoid dividing by

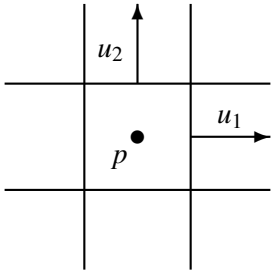

Fig. 10: Staggered grid cell in two-dimensional case

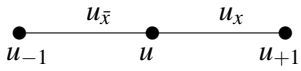

Fig. 11: Three point WENO stencil

zero.

6.2 Setting up Pressure Poisson Problem

Once the right-hand side values f_i, $i = 1,2,3$ are calculated, we can proceed to formulating pressure problem (4.4-4.5). Approximation of the right hand side in the Poisson equation (4.4) is straightforward. Proper approximation of the BC (4.5) is more involved [13].

Consider two-dimensional discrete continuity equation for accelerations $\dot{u}$ and $\dot{v}$ in the right boundary cell (N, j) on fig. 12:

$$(\dot{u}_{N,j} - \dot{u}_{N-1,j}) + (\dot{v}_{N,j} - \dot{v}_{N,j-1}) = 0 \tag{6.2}$$

where $\dot{u}_{N,j}$ is given on boundary Γ as a BC and other accelerations must be obtained from the discretized momentum equations:

$$\begin{aligned} \dot{u}_{N-1,j} &= f^1_{N-1,j} - \frac{p_{N,j} - p_{N-1,j}}{h} \\ \dot{v}_{N,j} &= f^2_{N,j} - \frac{p_{N,j+1} - p_{N,j}}{h} \\ \dot{v}_{N,j-1} &= f^2_{N,j-1} - \frac{p_{N,j} - p_{N,j-1}}{h} \end{aligned} \tag{6.3}$$

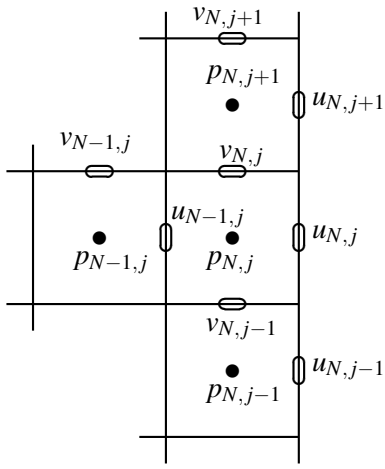

Fig. 12: Two-dimensional problem disctretization near the right boundary

Inserting the above accelerations into (6.2) and rearranging yields

$$\frac{3p_{N,j} - p_{N-1,j} - p_{N,j+1} - p_{N,j-1}}{h^2} = -\left(\frac{\dot{u}_{N,j} - f^1_{N-1,j}}{h} + \frac{f^2_{N,j} - f^2_{N,j-1}}{h} \right) \tag{6.4}$$

The Taylor expansion of (6.4) approximates boundary condition (4.5) applied on the right boundary:

$$\frac{\partial p}{\partial \mathbf{n}} = \mathbf{f} \cdot \mathbf{n} - \dot{\mathbf{u}} \cdot \mathbf{n} + O(h^2) \tag{6.5}$$

6.3 Semi-Implicit Bicolor Time Marching Algorithm

Consider Euler stepping procedure for discretization of evolution step (4.6) of marching process:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \tau(\mathbf{f^n} - \nabla p^{n+1}) \tag{6.6}$$

Any explicit marching procedure should satisfy the stability condition. This time step restriction takes convection, viscosity and gravity into account and ensures, that discrete information can evolve no further than one grid cell. This is necessary, since the discrete difference equations consider only fluxes between adjacent cells.

Let's define

$$C_i = \left(\left(\frac{\|u_i\|_\infty}{h} + \frac{1}{Re} \cdot \frac{6}{h^2} \right) + \sqrt{\left(\frac{\|u_i\|_\infty}{h} + \frac{1}{Re} \cdot \frac{6}{h^2} \right)^2 + \frac{|g_i|}{h}} \right)^{-1} \tag{6.7}$$

Values C_i in (6.7) are computed independently for each velocity component. The overall time step restriction is then expressed as

$$\tau \leq 2\xi \min(C_1, C_2, C_3) \tag{6.8}$$

where $\xi \in (0,1]$ is a safety factor.

In practice, however, due to accumulation of truncation errors, stability requires choosing ξ not exceeding 0.3. Resulting time step is quite short, so

the overall computing time is high. Usage of implicit time stepping could resolve the issue, but purely implicit solvers with good accuracy are very hard, if even possible, to implement with the level of parallelerism demanded for effective GPGPU computing.

However, a mathematically semi-implicit solver could be constructed, using the simmilar feature of the grid as RB-SSOR.

First, compute $\mathbf{f}(\mathbf{u}) = \mathbf{f}(\mathbf{u^n})$ for red nodes only. Then, compute $\mathbf{f}(\mathbf{u}) = \mathbf{f}(\mathbf{u^n} + \tau\mathbf{f}(\mathbf{u^n}))$ for black nodes. Even though the second formular is mathematically imlpicit, the actual computation is done in explicit manner, since only the most recent data from red nodes (which was obtained just before) is needed. In fact, this procedure could be rewritten as

$$\begin{aligned} \mathbf{f}_{\text{red}}(\mathbf{u}) &= \mathbf{f}_{\text{red}}(\mathbf{u}^n_{\text{black}}) \\ \mathbf{f}_{\text{black}}(\mathbf{u}) &= \mathbf{f}_{\text{black}}(\widetilde{\mathbf{u}}^{n+1}_{\text{black}}) = \mathbf{f}_{\text{black}}(\mathbf{u}^n_{\text{black}} + \tau\mathbf{f}_{\text{red}}(\mathbf{u}^n_{\text{black}})) \end{aligned} \tag{6.9}$$

This semi-implicit marching procedure allows to increase the safety factor for time step restriction due to better computational stability. This, in turn, results in a smaller overall amount of time steps. However, potential drawback of increasing the time step is a lower convergence rate of the Poisson solver on each time step, which is also decreased by computing right-hand side for black nodes with divergent velocity field. Also, for GPGPU computation, two kernel launches, instead of one for explicit scheme, will be needed for computing (6.9) correctly.

Depending on the specific problem properties, it could be whether beneficial in terms of overall computational time or not. However, the choice of

safety factor ξ is problem-dependent. Moreover, there is no formal procedure to determine it, so improved stability itself is worth of sacrificing some performance.

6.4 Implementation Details

Let's consider parallelerism of the described algorithm. Recall the marching procedure 5.

Algorithm 3 Marching process

1: **while** $t < T$ **do**

2: Compute $\mathbf{f} = -(\mathbf{u} \cdot \nabla \mathbf{u}) + \frac{1}{Re}\Delta \mathbf{u}$, using the velocity field from the previous time step $\mathbf{u}^n$

3: Solve the Poisson problem for pressure $\Delta p = \nabla \cdot \mathbf{f}$, $\nabla p \cdot \mathbf{n} = \frac{\partial p}{\partial \mathbf{n}} = \mathbf{f} \cdot \mathbf{n} - \frac{\partial \mathbf{u}}{\partial t} \cdot \mathbf{n}$

4: Compute divergence-free velocity field on the next time step from $\frac{\partial \mathbf{u}^{n+1}}{\partial t} = \mathbf{f} - \nabla p$ using some time discretization method.

5: $t = t + dt$

6: **end while**

Note, that on this coarse-grained level main loop of the algorithm 5 is sequential. First, accelerations for each velocity component are computed (step 2 in Algorithm 5).

Components of predicted acceleration field $\mathbf{f}$, f_1 and f_2, are computed independently. Since they use the same data about the velocity field on the

previous time step, they could be updated concurently in the single kernel.

Since Euler stepping is a fully explicit procedure, there is no global synchronization needed and only one kernel should be called.

However, for the Red/Black marching two kernel calls (for red and for black nodes) are necessary. In each of these calls, only half of the accelerations (one color) are computed, so it needs twice less threads for processing. Red color is denoted to nodes with $i+j+k$ even and black - to those with this index summ being odd.

Computation of corrected velocity field is done in explicit manner with $\frac{\partial \mathbf{u}^{n+1}}{\partial t} = \mathbf{f} - \nabla p$ and could be done in parallel in a single kernel call.

Consider predicted acceleration in two-dimensional case for x-direction (u and v - velocities in x and y directions correspondingly)

$$f_x = -u\frac{\partial u}{\partial x} - v\frac{\partial u}{\partial y} + \frac{1}{Re}\Delta u \tag{6.10}$$

To approximate the derivatives on the right-hand side we need velocity data from same neighbours as was needed for Poisson solver. For this reason, velocities are stored in the same way as pressure and simmilarily loaded to shared memory.

Finally, let's describe the data transfers between CPU and GPU during solver execution. Computational flow of the algorithm is sketched below.

After all memory allocations and ititialization of the problem, computation begins starting with $t = 0$:

Algorithm 4 Navier-Stokes Solver

1: **while** $t < T$ **do**
2: Compute time step $dt = 2\xi \min\{C_x, C_y\}$
3: Predict acceleration field $\mathbf{f}$ with Euler or Red/Black kernel
4: Compute right-hand side for Poisson equation $\varphi = \nabla \cdot \mathbf{f}$
5: Solve the Poisson problem for pressure with PCG alg. 2
6: Compute corrected velocity field, using pressure
7: $t \leftarrow t + dt$
8: **end while**

During algorithm 4 execution, there are following GPU-CPU transfers:

- Values C_x and C_y need to be transfered to CPU for computing time step. Since the flow of the algorithm is controlled by CPU, this couldn't be avoided
- Residual in PCG alg. 2 is transfered asynchronously on every Poisson iteration. This is again, algorithm control issue and couldn't be avoided
- Velocity and (if necessary) pressure field. GPU memory is not large enough for keeping the solutions on all time steps during the execution. They have to be transfered to CPU.

6.5 Verification

The classical lid-driven cavity problem has been investigated by many authors since some pioneer works giving good results of steady solutions twenty

years ago [21, 22]. Their results were confirmed by many other studies and the solution obtained at Re = 1000 for instance is quite close from one author to another [20, 24]. It became, in fact, a standard validation test for CFD algorithms.

Consider a square cavity with three fixed walls and a moving lid. At the initial moment $t = 0$, velocities are zero and the lid starts moving. Problem setting is sketched on fig. 13.

To numerically validate the GPU solver, the results were compared with the data from [21]. As is shown on fig. 14-15, the velocity components along the vertical and horizontal lines through the geometric center are in good agreement with results by Ghia [21] for both Reynolds numbers $Re = 100$ and $Re = 1000$.

6.6 Performance of Navier-Stokes Solver

Performance of the full INS solver was measured on a lid-driven cavity problem, described in the previous subsection.

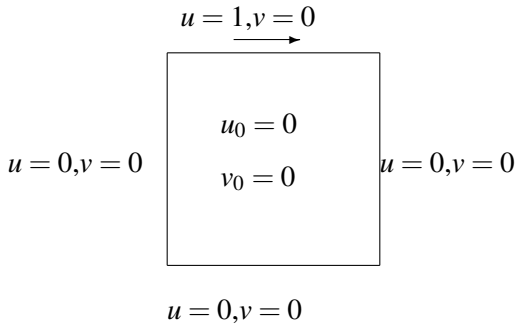

Fig. 13: Sketch of the lid-driven cavity problem formulation

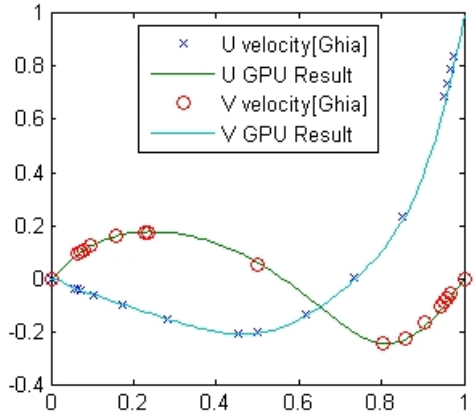

Fig. 14: Numerical data verification, $Re = 100$

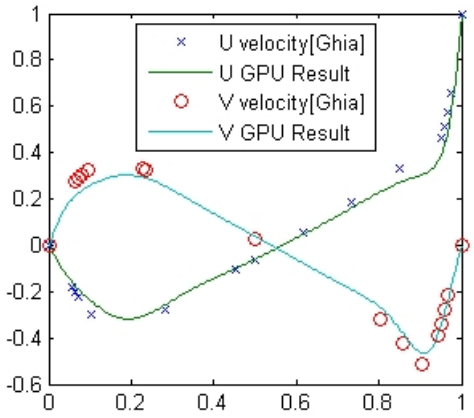

Fig. 15: Numerical data verification, $Re = 1000$

Computation was done both for the steady-state and highly unsteady solutions. In case of lid-driven cavity problem, steadiness of the solution completely depends on the length of time domain. For the small computing times velocity field evolvs rapidly, which allows us to measure solver performance on rapidly changing type of problems.

On the contrary, for relatively large time domains, steady solutions dominate.

Numerical experiments were run for two-dimensional problem with $Re = 1000$ on a grid with size 512×512. The performance measurements are summarized on the fig. 16. There, Euler-03 denotes computations with ordinary explicit Euler stepping without RB-SSOR preconditioning for Poisson and safety factor $\xi = 0.3$, Euler-03-SSOR stands for the same solver with RB-SSOR-preconditioned Poisson solver. Analogously, RB-08-SSOR denotes semi-implicit bicolor time marching with RB-SSOR-preconditioned Poisson solver and a safety factor $\xi = 0.8$.

As seen on fig. 16, for both time marching processes preconditioning of Poisson problem is beneficial, especially for rapidly changing problem. While the solution is approaching steady state, the gain from preconditioning is getting lower.

This effect is due to the following: the smaller the velocity changes are in the ”prediction” step, the less ”correction” iterations (Poisson iterations in this case) are necessary. So, for steady solutions a very low number of Poisson iterations is needed and the preconditioning becomes a pure waste of extra computational power. The only reason, why it is still beneficial is extensive use of the Poisson solver in the initial unsteady region. Overall

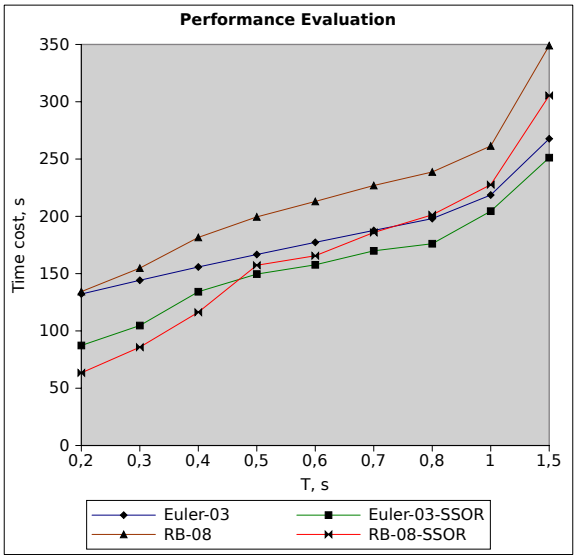

Fig. 16: Performance comparison

cost is decomposed on fig. 17 to demonstrate the use of Poisson solver in steady and unsteady regions.

The developed bicolor time marching procedure without preconditioning actually decreases performance, even though the time step was increased. This is due to the following: with increased time step and extra diffusion an extensive use of Poisson solver is necessary, even for the late steady stages. That is why preconditioning had such a significant impact on bicolor marching.

Red/Black time stepping with preconditioned Poisson solver was found to be beneficial only for highly unsteady solutions. However, as mentioned before, it allows more freedom for choice of safety factor, which could be important as well, since there is no formal procedure to determine it and

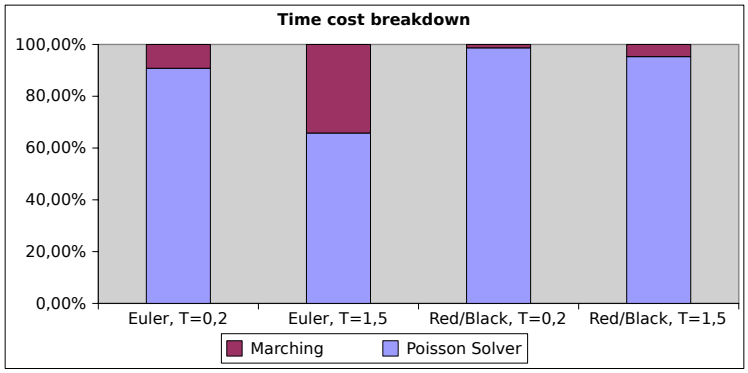

Fig. 17: Time cost breakdown

thus the choice is left to engineer.

제 7 장

Mixed Precision Modification

From one hand, due to high accuracy demands, computation should be done with the double precision. Moreover, purely single precision solver gives a poor convergence for the Poisson problem. Due to restriction on maximum amount of Poisson iterations, for some time steps it doesn't cancel out the error complitely, which introduces the divergence to the velocity field. The solution obtained this way is not divergence-free and is, therefore, non-physical. With certain amount of introduced divergence, the whole algorithm will not converge. All these factors lead to choice of double precision as an only option for algorithm.

From the other hand, originally, GPUs suppoted single precision computations only. Even though last several generations support double precision, it is still much slower than handling single precision floats. Hence, it is desired to do as much computation in single precision as it is possible.

Considering both arguements, a compromise could be found in developing a mixed precision solver. It has to satisfy the following demands:

1. The most expensive operations should be carried out in single precision.

2. Operations, having the biggest impact on convergence, should be done in double precision.

3. Result of each time step should be simmilar to the one carried completely in double precision.

7.1 Iterative Refinement

Recall the general description of the marching algorithm:

Algorithm 5 Marching process

1: **while** $t < T$ **do**

2: Compute $\mathbf{f} = -(\mathbf{u} \cdot \nabla \mathbf{u}) + \frac{1}{Re}\Delta \mathbf{u}$, using the velocity field from the previous time step $\mathbf{u}^n$

3: Solve the Poisson problem for pressure $\Delta p = \nabla \cdot \mathbf{f}$, $\nabla p \cdot \mathbf{n} = \frac{\partial p}{\partial \mathbf{n}} = \mathbf{f} \cdot \mathbf{n} - \frac{\partial \mathbf{u}}{\partial t} \cdot \mathbf{n}$

4: Compute divergence-free velocity field on the next time step from $\frac{\partial \mathbf{u}^{n+1}}{\partial t} = \mathbf{f} - \nabla p$ using some time discretization method.

5: $t \leftarrow t + dt$

6: **end while**

Steps 2 and 4 and third task are done only once in a time step and are of crucial importance for convergence. These operations are not available for lowering the precision.

The Poisson solver dominates the overall computation, which satisfies the demand 1. Inner iterations of Poisson solver don't affect the overall velocity field convergence (demand 2), if the final overall solution of pressure problem is simmilar to the one computed with double precision (demand 3).

Thus, computations in the Poisson solver could be partly simplified to single precision.

Consider iterative refinement procedure with explicit PCG restarts. It is summarized in algorithm 6.

Algorithm 6 Iterative refinement

while $l < l_{max}$ and $r^{\mathrm{d}}_{old} > \varepsilon^{\mathrm{d}}$ **do**
  $r^{\mathrm{d}}_{l} \leftarrow \varphi^{\mathrm{d}} - A p^{\mathrm{d}}_{l}$
  $h^{\mathrm{d}} \leftarrow M^{-1} r^{\mathrm{d}}_{l}$
  $d^{\mathrm{f}} \leftarrow h^{\mathrm{f}}$
  $r^{\mathrm{d}}_{old} \leftarrow (r^{\mathrm{d}}_{l}, h^{\mathrm{d}})$, $i \leftarrow 0$
  while $i < i_{\mathrm{restart}}$ and $r^{\mathrm{f}}_{old} > \varepsilon^{\mathrm{f}}$ **do**
    $t^{\mathrm{f}} \leftarrow A d^{\mathrm{f}}$
    $\alpha \leftarrow \dfrac{r^{\mathrm{f}}_{old}}{(d^{\mathrm{f}}, t)}$
    $r^{\mathrm{f}}_{l+1} \leftarrow r^{\mathrm{f}}_{l} - \alpha t^{\mathrm{f}}$
    $p^{\mathrm{f}}_{l+1} \leftarrow p^{\mathrm{f}}_{l} + \alpha d^{\mathrm{f}}$
    $h^{\mathrm{f}} \leftarrow M^{-1} r^{\mathrm{f}}_{l+1}$
    $r^{\mathrm{f}}_{new} \leftarrow (r^{\mathrm{f}}_{l+1}, h^{\mathrm{f}})$
    $\beta \leftarrow \dfrac{r^{\mathrm{f}}_{new}}{r^{\mathrm{f}}_{old}}$
    $d^{\mathrm{f}}_{l+1} \leftarrow h^{\mathrm{f}} + \beta d^{\mathrm{f}}_{l}$
    $r^{\mathrm{f}}_{old} \leftarrow r^{\mathrm{f}}_{new}$, $i \leftarrow i+1$, $l \leftarrow l+1$
  end while
end while

Upper indices ”f” and ”d” denote values in single and double precision

respectively.

The computations in outer loop (2-5) are carried out in double precision, which makes the overall solution satisfy the equivalence demand 3. On the contrary, inner loop (7-15), which dominates the computation, is repeated in single precision - satisfying the performance bottleneck demand 2.

Best performance results for double precision solver were compared to the corresponding results of the mixed iterative refinement solver (see fig. 18).

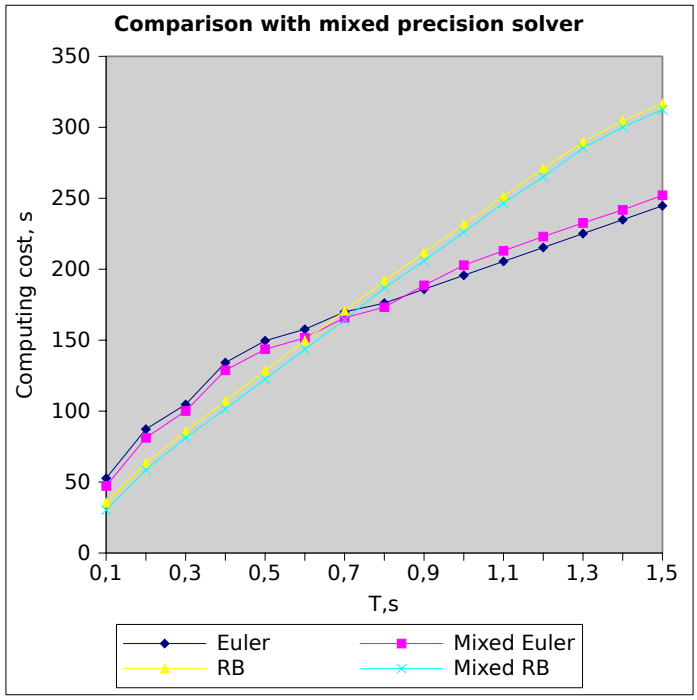

Fig. 18: Performance comparison of double and mixed precision solvers

Upgrade in performance due to the use of mixed precision solver doesn't

exceed 5% in the highly unsteady phase, decreasing with further time marching to the steady region. In fact, for Euler marching it even increases the overall computation cost of obtaining steady solution. Explanation of this performance drawback in steady region is provided below.

The mixed precision Poisson solver needs more iterations to converge because of higher round-off errors. For rapidly changing velocity field the amount of Poisson iterations is high for both double and mixed precision solvers and poorer convergence is outweighted by higher average speed of floating point operations. On the contrary, in the steady region only several Poisson iterations are needed for convergence of double precision solver, while the mixed precision one increases this amount to tens of them. Hence, the outcome of tradeoff between number of iterations and individual iteration cost is opposite to the one above and the gain from a more complicated mixed precision solver becomes neglectable and even negative.

7.2 Further Elaboration

Performance of the mixed precision algorithm 6 could be further refined. In order to do that, we will use a heuristic approach inspired by work [23]. In this work, fixed point processing is considered for optimization with choosing an appropriate word length for each part of the system. Simmilarily, we will consider choice of single or double precision for each part of the solver.

Consider inner loop operations (7-15) in algorithm 6. Each of them could be carried whether in single or double precision. We will investigate an impact of doing each of them in single precision, while others are com-

puted with double precision. Thus, we can obtain an impact of lowering precision for an individual operation on overall computing cost.

Simulation was run for a Poisson solver with right-hand side corresponding to unsteady and steady region. Resulting impact on the overall time cost is summarized in the table 1.

표 1: Impact of lowering precision

Operation carried in single precision	Performance Gain
$t \leftarrow Ad$	$+4\%$
$\alpha \leftarrow (d,t)$	$+2\%$
$\alpha \leftarrow r_{old}/\alpha$	-3%
$p \leftarrow p+\alpha d$	0%
$r \leftarrow r-\alpha t$	0%
$h \leftarrow M^{-1}r$	$+4\%$
$r_{new} \leftarrow (r,h)$	$+2\%$
$\beta \leftarrow r_{new}/r_{old}$	-3%
$d \leftarrow h+\beta d$	-1%

It could be noted from table 1, that lowering the precision of different operations has a different impact on overall cost. Precision should be lowered only for operations, which impact on convergence is positive, and kept high for those with negative and zero effect. Practically, only matrix-vector and dot product operations should be carried in single precision.

Performance of the resulting mixed precision solver is shown on fig. 19.

For Euler stepping, performance is further upgraded (compared to previously discussed mixed precision solver) in unsteady region. This performance gain is around 3% as compared to first mixed solver. On the other hand, in steady region, where the previous mixed precision modification

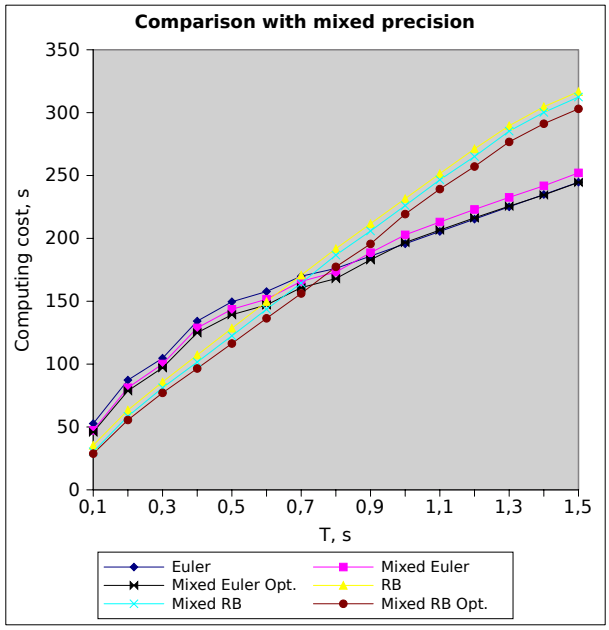

Fig. 19: Performance of the optimized mixed precision solver

was giving a negative effect, current solver shows simmilar performance as double precision solver.

For Red/Black marching, performance is further refined in all simulation domain. As compared to algorithm 6, gain is varying from 5% in unsteady region to 3% in steady part.

제 8 장

Unsteady flows

Tools for simulation of unsteady flows are crucial for many engineering applications. They include variety of start up processes, such as cold start of a turbine or beginning of flow in pipes. These processes have both steady and unsteady parts.

Turbulence is another unsteady flow effect. Most flows of interest, such as fluid transfer in a pipe with a pulp, are turbulent in nature and don't converge to any steady state.

Performance of developed marching process in comparison with Euler marching was measured on a lid-driven cavity problem. It is known [24], that with Reynolds number close to $Re = 10000$ flow in a lid-driven cavity becomes turbulent. Result of numerical experiments with $Re = 10000$ are shown on the fig. 20. The relative speedup compared to Euler marching was varying from 13% to 30%, depending on size of time domain.

Illustrated performance was compared to sequential CPU code. Resulting speedup is shown on fig. 21.

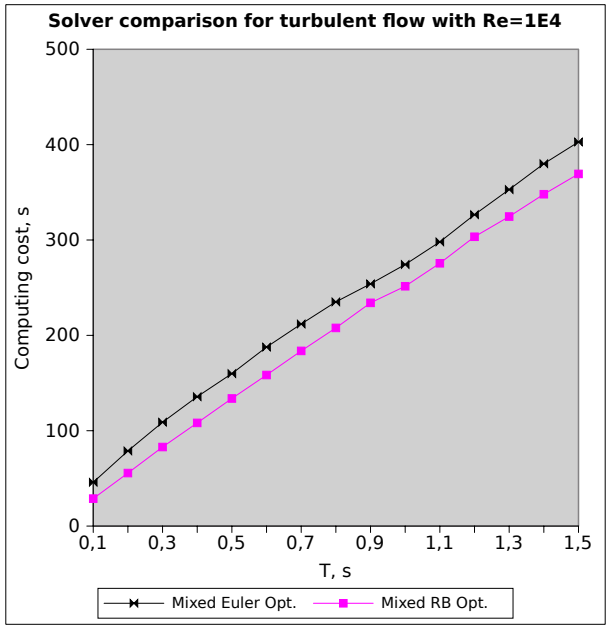

Fig. 20: Performance comparison for $Re = 10000$

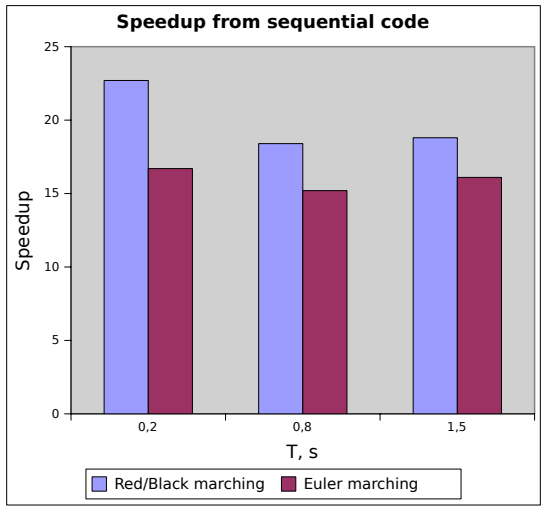

Fig. 21: Speedup relative to CPU code ($Re = 10000$)

제 9 장

Summary

Projection method was used to develop a GPGPU solver for incompressible Navier Stokes equations.

As a Poisson solver for inner iterations, matrix-free version Preconditioned Conjugate Gradient method was chosen. Several existing matrix-free preconditioners were considered, including recently introduced Incomplete Poisson preconditioner [10]. After the tests, Red/Black SSOR, which is mostly used as a separate solver, was chosen.

Secondly, time marching process was considered. A semi-implicit bi-color marching process, inspired by Red/Black scheme was developed. It was shown to be beneficial for rapidly changing flows by measuring solver's performance on a turbulent problem, compared to explicit Euler marching technique.

Finally, Poisson solver was considered for further optimization with introducing mixed precision computation. Impact of lowering the precision of individual operations was measured. Based on this information, a mixed precision solver was developed.

The final version of the solver showed speedup of up to 22 as compared with sequential CPU code performance (fig. 21).

참고 문헌

[1] J. Stam, - ”Stable Fluids”, SIGGRAPH, 1999, pp 121-128

[2] M.J. Harris, W.V. Baxter, T. Scheuermann, A. Lastra, - ”Simulation of cloud dynamics on graphics hardware”, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003, pp 92–101

[3] Y. Liu, X. Liu, E. Wu, - ”Real-time 3D fluid simulation on GPU with complex obstacles”, 12th Pacific Conference on Computer Graphics and Applications, 2004, pp 247–256

[4] T. Brandvik, G. Pullan, - ”Acceleration of a 3D Euler solver using commodity graphics hardware”, 46th AIAA Aerospace Sciences Meeting and Exhibit, 2008

[5] E. Elsen, P. LeGresley, E. Darve, - ”Large calculation of the flow over a hypersonic vehicle using a GPU”, Journal of Computational Physics 227 (2008), pp 10148–10161

[6] J. Liu, Z. Ma, S. Li, Y. Zhao, - ”A GPU Accelerated Red-Black SOR Algorithm for Computational Fluid Dynamics Problems”

[7] M. Griebel, P. Zaspel, - ”A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations”, Comput Sci Res Dev 25 (2010), pp 65–73

[8] J.C. Thibault, I. Senocak, - ”CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows”, 47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition, 2009

[9] E. Konstantinidis, Y. Cotronis, - ”Graphics processing unit acceleration of the red/black SOR method”, Concurrency Computatation: Practice Experience (2012)

[10] M. Ament, G. Knittel, D. Weiskopf, W. Strasser, - ”A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform”, Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference, pp 583 - 592

[11] R. Helfenstein, J. Koko, - ”Parallel preconditioned conjugate gradient algorithm on GPU”, Journal of Computational and Applied Mathematics, Vol. 236, Issue 15, Sep. 2012, p. 3584–3590

[12] A.J. Chorin, - ”Numerical Solution of the Navier-Stokes Equations”, Math. Comp. Vol. 22, pp 745–762

[13] P.M. Gresho, R.L. Sani, - ”On Pressure Boundary Conditions for the Incompressible Navier-Stokes Equations”, Int. J. for Numerical Methods in Fluids, Vol.7, pp 1111-1145

[14] Y. Saad, - ”Iterative methods for sparse linear systems”, 2nd edition, SIAM, Philadelpha, PA, 2003

[15] R. Li, Y. Saad, - ”GPU-Accelerated Preconditioned Iterative Linear Solvers”, The Journal of Supercomputing, Vol. 63, Issue 2, Feb. 2013, pp 443-466

[16] J.M. Elble, N.V. Sahinidis, P.Vouzis, - ”GPU computing with Kaczmarz’s and other iterative algorithms for linear systems”, Parallel Computing 36 (2010), pp 215–231

[17] L. Li, L. Li and Y. Guangwen, - ”A Highly Efficient GPU-CPU Hybrid Parallel Implementation of Sparse LU Factorization”, Chinese Journal of Electronics, Vol.21, No.1, Jan. 2012

[18] N. Galoppo, N.K. Govindaraju, M. Henson, D. Manocha, - ”LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware”, Proceedings of the 2005 ACM/IEEE conference on Supercomputing

[19] X.-D. Liu, S. Osher and T. Chan, - ”Weighted essentially non-oscillatory schemes”, Journal of Computational Physics, Vol. 115, pp 200-212, 1994

[20] C.-H. Bruneau, M. Saad, - ”The 2D lid-driven cavity problem revisited”, Computers & Fluids, Vol. 35, pp 326-348 (2006)

[21] U. Ghia, K.N. Ghia, C.T. Shin, - ”High-resolutions for incompressible flows using Navier–Stokes equations and a multigrid method”, Journal of Computational Physics, Vol.48 (1982)

[22] R. Schreiber, H.B. Keller, - ”Driven cavity flows by efficient numerical techniques”, Journal of Computational Physics, Vol. 49 (1983)

[23] Wonyong Sung, Ki-Il Kim, - ”Simulation-based word-length optimization method for fixed-point digital signal processing systems”, IEEE Transactions on Signal Processing Vol. 42, Is. 12, pp 3087-3090

[24] P. N. Shankar, M. D. Deshpande, - ”Fluid Mechanics in the Driven Cavity”, Annual Reviews in Fluid Mechanics, Vol. 32, pp 93-136 (2000)

초 록

푸아송 방정식을 위한 레드/블랙 접근 방법에서 착안된 새로운 반-내포의 이색 나 비에 스토크스 방정식이 GPGPU를 활용한 비압축성 유체 시뮬레이션을 위해 연 구되었다. 이 방법은 극도로 불안정한 유체 흐름을 시뮬레이션 하는 경우에 특히 효과적이라는 것이 입증된 바 있다. 또한 이 방법은 안정조건에 여유가 있기 때문 에, 시간 간격을 선택하는데 있어 더 큰 자유도를 제공한다는 장점이 있다. 공간 적 이산은 엇갈림격자를 사용하고, 대류에 의해 결정되는 흐름을 관찰하기 위해 WENO 방법을 사용하여 대류항을 근사한다. 이 연산에 지배적인 영향을 주는 것 은 행렬을 사용하지 않는 전처리된 켤레구배법(Preconditioned Conjugate Gradient) 알고리즘을 사용하는 푸아송 방정식 풀이 방식이다. 이미 알려진 행렬 을 사용하지 않는 여러 전처리 조절방법들이 고려되었다. 레드/블랙 SSOR은 일반 적으로 주 풀이방법으로 사용되지만,행렬을 사용하지 않는 PCG의 전처리 조절방 법 중 가장 적합한 방법인 것으로 밝혀졌다. 명시적인 PCG의 재시작을 이용한 혼 합 정밀도 조정은 불안정한 구간에서 약간의 이득을 주지만, 안정적인 구간에서는 불이익을 준다. 보다 나은 성능의 최적화를 위해 풀이 방법의 감도에 대한 연구가 수행되었다.

주요어 : 비압축성 나비에-스토크스 방정식, 레드/블랙 접근 방법,불특정한 유체 흐름,엇갈림격자, WENO, 푸아송 방정식,레드/블랙 SSOR, 혼합 정밀도

학번 : 2011-24073